

ESD-TDR-64-161

SR-122

## PROGRAM STRUCTURE FOR MILITARY REAL-TIME SYSTEMS

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-161

JANUARY 1965

## ESD RECORD COPY

RETURN TO  
SCIENTIFIC & TECHNICAL INFORMATION DIVISION  
(ESTI), BUILDING 1211

COPY NR. \_\_\_\_\_ OF \_\_\_\_\_ COPIES

J. H. Burrows

Prepared for

DIRECTORATE OF COMPUTERS  
ELECTRONIC SYSTEMS DIVISION  
AIR FORCE SYSTEMS COMMAND  
UNITED STATES AIR FORCE

L. G. Hanscom Field, Bedford, Massachusetts



Projects 416 and 502

Prepared by

THE MITRE CORPORATION  
Bedford, Massachusetts  
Contract AF19(628)-2390

ESTI PROCESSED☐ DDC TAB ☐ PROJ OFFICER☐ ACCESSION MASTER FILE☐ \_\_\_\_\_

DATE \_\_\_\_\_

ESTI CONTROL NR **AL** 44484

CY NR \_\_\_\_\_ OF \_\_\_\_\_ CYS

ADJ0610318

Copies available at Office of Technical Services,  
Department of Commerce.

Qualified requesters may obtain copies from DDC.  
Orders will be expedited if placed through the librarian  
or other person designated to request documents  
from DDC.

When US Government drawings, specifications, or  
other data are used for any purpose other than a  
definitely related government procurement operation,  
the government thereby incurs no responsibility  
nor any obligation whatsoever; and the fact that the  
government may have formulated, furnished, or in  
any way supplied the said drawings, specifications,  
or other data is not to be regarded by implication  
or otherwise, as in any manner licensing the holder  
or any other person or corporation, or conveying  
any rights or permission to manufacture, use, or sell  
any patented invention that may in any way be related  
thereto.<sup>4</sup>

Do not return this copy. Retain or destroy.

# PROGRAM STRUCTURE FOR MILITARY REAL-TIME SYSTEMS

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-161

JANUARY 1965

J. H. Burrows

Prepared for

DIRECTORATE OF COMPUTERS  
ELECTRONIC SYSTEMS DIVISION  
AIR FORCE SYSTEMS COMMAND  
UNITED STATES AIR FORCE

L. G. Hanscom Field, Bedford, Massachusetts



Projects 416 and 502

Prepared by

THE MITRE CORPORATION  
Bedford, Massachusetts  
Contract AF19(628)-2390

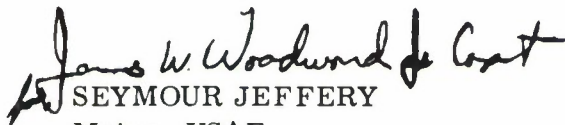
## PROGRAM STRUCTURE FOR MILITARY REAL-TIME SYSTEMS

### ABSTRACT

This report describes the program design and accompanying tools required to fabricate an initial operational program which is subject to change through use and as a result of changes in the environment of the system. Major design decisions are shown to have little relation to the programming language that is chosen for the statement of the logical data transformation to be implemented by the final program. The actual coding required for performing the logical transformations turns out to be a relatively small part of the overall design job. The choice of the programming language to be used is just an initial decision in the process of selecting adequate tools for assisting the production and modification of real-time programs.

### REVIEW AND APPROVAL

This technical documentary report has been reviewed and is approved.

The image shows a handwritten signature in dark ink, which appears to read "James W. Woodward for Co. 1". The signature is written in a cursive, somewhat stylized script.

SEYMOUR JEFFERY

Major, USAF

Chief, Computer Division

Directorate of Computers

Deputy for Engineering & Technology

## CONTENTS

<u>Section</u>		<u>Page</u>
I	INTRODUCTION	1
II	TASK AREAS	3
III	IMPLICATIONS OF PROGRAM-DESIGN DECISION	6
IV	CONSTRUCTION TOOLS USED FOR ON-LINE SYSTEMS	15
V	CONCLUSIONS	22

## ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
1	Tasks or Subprograms	3
2	Internally Associated Items	5
3	Subprogram Tables	7
4	Estimate of File Size	7
5	Instruction Requirements	8
6	Word Requirements	8
7	Process Sequence	9
8	Dynamic Memory Allocation	12
9	First Stage of Translation--Compiling	17
10	Second Stage of Translation--Loading	18
11	Simulation-Data Production	19
12	Debug-Run Flow	20
13	Recorded-Data Translation	20
14	Assembly Testing	21

# PROGRAM STRUCTURE FOR MILITARY REAL-TIME SYSTEMS

## SECTION I

### INTRODUCTION

At the present time a great deal of attention is being devoted to programming languages and their relative effectiveness in performing the logical transformations required in data-processing tasks. In fact, so much emphasis is being given to such languages that there is a danger that program design will be construed as being merely a matter of choosing and applying the appropriate programming language for each job. Actually, there is much more involved.

One of the most important problems in program design is that of logistics. The data must be made available to the machine at the right time and in the right sequence. It is a fact that, for a large problem, the logistics involved in the machine processing of data becomes much greater and much more critical than the logical problem of transformation of data. There are many mathematicians who can tell you in what sequence to add certain numbers together and what logical processes to carry out to get a certain answer. But making a computer accomplish a desired data manipulation is much more complicated. There are, for example, the problems of getting the raw data into the machine, of storing partially finished calculations, and of making the machine perform the required data transformations. Furthermore, in the operation of a data-processing job, the machine's memory gets loaded with programs, and, when you have a large, complex job, these programs won't all fit in the storage space available. This is one of the most serious problems encountered in machine problem-solving.

It is characteristic of all the significant problems associated with the military that the whole program never fits the machine. Therefore, we must somehow break up the job into individual tasks. When we have done this, we must determine what data these tasks require and what data they produce, structure the data in some way compatible with them, determine the sequence of running the tasks, and allocate storage. Then we have not only the problem of allocating the main computer storage, but also the problems of storing the intermediate results, the data not yet required, and the subprograms not in use, and of scheduling the flow for the total processing.



## SECTION II

### TASK AREAS

To illustrate this point, a hypothetical programming-design job is broken into six tasks areas, labelled A-F in Fig. 1. Let us, for the moment, skip the problem of how to divide an overall processing task into subprograms. For one thing, the rationale for such a split will not be entirely clear until we have seen the effects that different decisions at this point have upon later decisions. Furthermore, each breakdown depends upon the content of the total processing required, which, for this report, is discussed in abstract terms only. The point here is that, no matter how we divide the overall processing job into subprograms, such a breakdown is necessarily a vital step in developing a design that will enable us to proceed with the programming of the individual tasks.

INFORMATION ITEMS	A	B	C	D	E	F
	1 ✓	✓				
	2			✓	✓	
	3 ✓	✓				
	4		✓		✓	
	5 ✓				✓	✓
	6		✓			
	7			✓	✓	
	8 ✓		✓			
	9	✓		✓		
	10			✓		✓
	11	✓	✓			
	12 ✓	✓			✓	

Fig. 1 Tasks or Subprograms



Let us now consider the data elements that are to be processed in our example. In addition to knowing what data elements are involved in the total job, it is essential to know which are needed and/or produced by each task area. The matrix of information items for the present example is shown in Fig. 1.

The information items are grouped on the basis of natural association by the point of view outside the computer. For example, Items 1-4 might be information about airfields. Therefore, although these items may be stored in separate parts of the computer memory, they are logically associated from the outside point of view. In their operation, the individual tasks either require or produce the information items checked in Fig. 1. Therefore, while each subprogram is running, the data associated with it must be in the main computer store, for if information is needed, it must be available, and if it is to be produced, there must be local space saved for it. The matrix shown in Fig. 1 represents a relatively simple program. In actual programs the picture is often much bigger. In the SAGE program breakdown, for example, there were some 70 task areas and over 2,000 data elements.

The information in the matrix has several interesting features. For instance, we can see that Information Items 1 and 3 are needed by Subprograms A and B but not by C, D, E, or F. So, internally, Items 1 and 3 are associated. In other words, from an internal point of view, all of the information items required or produced by a task area are associated. We might want to group these internally associated items of information and treat them as a unit. That is, we might want to arrange it so that, whenever we move Item 1 into or out of the main computer storage, we shall also move in or out Item 3. For example, we might wish to group the internally associated items as shown in Fig. 2. Here we see Items 1 and 3 grouped together as Table I, Items 2 and 4 as Table II, Items 5 and 6 as Table III, Items 8 and 9 as Table IV, Item 10 as Table V, and Items 11 and 12 as Table VI.

INFORMATION ITEMS	A	B	C	D	E	F
	1 ✓	✓				
	2			✓	✓	
	3 ✓	✓				
	4		✓		✓	
	5 ✓				✓	✓
	6		✓			
	7			✓	✓	
	8 ✓		✓			
	9	✓		✓		
	10			✓		✓
	11	✓	✓			
	12 ✓	✓			✓	

TABLES

I = 1,3

II = 2,4

III = 5,6,7

IV = 8,9

V = 10

VI = 11,12

Fig. 2 Internally Associated Items

### SECTION III

#### IMPLICATIONS OF PROGRAM-DESIGN DECISION

Let us consider the implications of this kind of program-design decision. In place of the items of information, we now have tables or files, as shown in Fig. 3. We need Tables I, III and IV for A, Tables I, IV, and VI for B, etc. Now we have some insight into the table or file structure and the program structure for the total processing job, and we can make some estimate of file sizes, as shown in Fig. 4, if the user provides, for example, data giving the number of airfields he wishes to talk about. Knowing what these sizes are, we can calculate that when Subprogram A runs, the data storage area alone that is required is 4,000 plus 1,000 plus 2,000, or a total of 7,000 registers of main memory. Now, if we know how many instructions are required for each task area as given in Fig. 5, we can get a better idea of the total memory requirement for each subprogram. We can estimate fairly accurately what the total storage requirement for each task will be.

We can see in Fig. 6 that the subprogram requiring the greatest number of words is Task D, which requires 11,000 words. Now suppose that the available machine has only 10,000 registers of memory. Obviously, more design work must be done. It might be appropriate to keep the program the same size and subdivide the data, and plan on bringing the data past this program piece by piece. Or, we might split Subprogram D into two equal parts, D, and D'. Or, some of the small jobs of Task D might be moved out of that task and into another, such as Task E, which requires only 6,000 memory registers. Or, we could change the table sizes by splitting them up into pieces, half size or even smaller. There are many possible solutions to the problem. It is true of all of them that, in addition to changing the subprogram in question, they will usually affect other task areas as well.

TABLES OR FILES		SUBPROGRAMS					
		A	B	C	D	E	F
	I	✓	✓				
	II			✓	✓	✓	
	III	✓		✓	✓	✓	✓
	IV	✓	✓	✓	✓		
	V				✓		✓
	VI		✓	✓		✓	

Fig. 3 Subprogram Tables

TABLES OR FILES		SUBPROGRAMS						SIZE OF DATA
		A	B	C	D	E	F	
	I	✓	✓					4000
	II			✓	✓	✓		2000
	III	✓		✓	✓	✓	✓	1000
	IV	✓	✓	✓	✓			2000
	V				✓		✓	4000
	VI		✓	✓		✓		1000

Fig. 4 Estimate of File Sizes

		NUMBER OF INSTRUCTIONS						SIZE OF DATA
		2000	1500	1500	2000	2000	1500	
		SUBPROGRAMS						
		A	B	C	D	E	F	
TABLES OR FILES	I	✓	✓					4000
	II			✓	✓	✓		2000
	III	✓		✓	✓	✓	✓	1000
	IV	✓	✓	✓	✓			2000
	V				✓		✓	4000
	VI		✓	✓		✓		1000

Fig. 5 Instruction Requirements

		NUMBER OF INSTRUCTIONS						SIZE OF DATA
		2000	1500	1500	2000	2000	1500	
		SUBPROGRAMS						
		A	B	C	D	E	F	
TABLES OR FILES	I	✓	✓					4000
	II			✓	✓	✓		2000
	III	✓		✓	✓	✓	✓	1000
	IV	✓	✓	✓	✓			2000
	V				✓		✓	4000
	VI		✓	✓		✓		1000
		9K	8.5K	7.5K	11K	6K	6.5K	
WORDS								

Fig. 6 Word Requirements



To simplify, let us suppose that the machine available has 12,000 registers in main memory. Thus far there is no design conflict among subprograms. Now, in order to proceed further in the overall design, we must consider the sequencing of the individual tasks. Suppose that Task A is an input program which has to operate before anything else, and that Task F is an output program which must be the last subprogram processed. For the other tasks there are only the following types of constraints: Some tasks in B must precede D -- there is nothing in D that must precede B. If it had turned out that there were some things to be done in Task D that were logically required before B, and some to be done in D that were logically required after B, there would have been a problem, and the task breakdown would have had to be redone. In that case, we might split D into two parts - one part that precedes B and one that follows B. Suppose that such is not the case, however, and that the precedence relations shown at the top of Fig. 7 are the logical constraints. We see that the



#### POSSIBLE CONTROL SEQUENCES

A	B	C	D	E	F
A	B	C	E	D	F
A	C	B	D	E	F
A	C	B	E	D	F
A	C	E	B	D	F

Fig. 7 Process Sequence

following rules of order prevail: A must precede all other tasks, F follows all others, B precedes D, and C precedes D and E. Given these four rules of order, we can derive five possible control sequences, which are shown in the lower half of Fig. 7. These five sequences may not all be equivalent; they may have different running times on the computer, or one of these sequences may be simpler and better for checkout, or one may be better for extension in the future. These differences have to be considered by the program designer.

Suppose that we just want to investigate the first sequence - A, B, C, D, E, F. Once we know the operating sequence of the task areas, the data requirements, and size of each task area, it is appropriate to do a preliminary storage, or resource, allocation. We shall consider only one resource, namely main memory, although such things as secondary storage and I/O channels must also be allocated and scheduled for use.

Main memory allocations can be viewed as a two-dimensional plot, with time as one axis and memory cells as the other axis. For each subprogram time slot, an allocation of main memory to its program and data must be chosen. For the example that we have been following, we already know that we can do this since there are 12K memory cells available and there is no subprogram that requires, at least at this stage of design, more than 11K. Some of the guides for storage allocation in our example will be:

- (1) Internally associated data will be kept as a unit (called contiguous addressing) when in the main store.
- (2) An attempt will be made to move data as little as possible consistent with use and storage limitations.
- (3) If a data table is used by several programs, an attempt will be made to have it occupy the same addressed area of memory for all of its uses.



- (4) Where possible, data transfers will be overlapped with program operations; i. e., transfers preparing for future program operation will take place during the operation of any program.

These rules are only for guiding the solution of the simple problem presented by the example discussed here. Different rules are followed for different systems, depending upon the complexity of the job, the time requirements, and the translation and control machinery available for stating or executing the design. In any case, the ultimate goal of main-memory allocation is to minimize the amount of nonuseful setup time for each operating program while keeping the overhead on-line structure both small and short in operating time. In order to do this, the allocation must be easy to modify and must allow debugging to proceed, i. e., additional tools may be needed for each level of sophistication in a storage allocation. A truly dynamic allocation will require different translators and larger amounts of overhead machinery on-line, especially for debugging, operational testing, monitoring, and recovery.

To return to the sample program, one of the many possible memory layouts is given in Fig. 8. The size of the overlay problem in this example is small. The total system is estimated (see Fig. 6) to require some 24.5K registers, and the main memory available is 12K, resulting in an overlay factor of only 2:1. The original SAGE program, on the other hand, had an overlay factor of about 15:1 with a main memory of 8K words.

Figure 8 shows that there are several tables that do not move during program operation. In an operational military system, portions of the data may sometimes have to be saved after program operation in order to allow recovery should normal operation become interrupted, e. g., because of power failure, program error, machine error, or scheduled stoppage. Thus, for those considerations this design may be inadequate. In the design presented here,

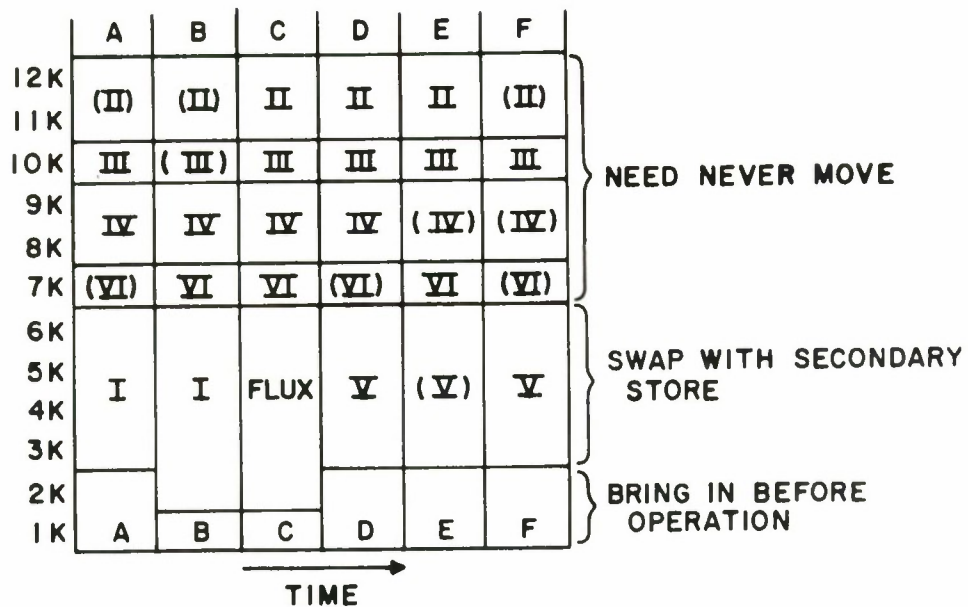


Fig. 8 Dynamic Memory Allocation

however, it is assumed that it is not necessary to save a program after its operation. It is assumed that programs create their cycle-to-cycle memory in isolable tables of the system and thus have no persistent internal memory. Because of this characteristic, normal initialization procedures may be imbedded in the steady state of the code and therefore are not required at the start of each operation.

Having settled on a storage allocation, we must next derive an I/O schedule that will accomplish the program transfers required for creating the appropriate environment for each program unit. For the example given here, this involves transfer immediately prior to operation, and the alternate saving and restoration of Tables I and V in the appropriate storage area. In more realistic systems, such control of I/O and the sequencing operation is done by a central executive that may have many more duties than that of program-environment control.

It is at this point that the program designer can attempt to estimate the running time of a cycle of the operational program for the particular machine configuration he is concerned with. However, even at this point in program design, obtaining accurate size estimates of the subprograms involved is a major problem, and estimating the running time of subprograms whose most frequently operated instruction next to a load accumulator is a conditional branch, is extremely difficult. Therefore, it is evident that final design decisions must be delayed until later in the fabrication cycle. Tools, procedures, and early design decisions giving the gross structure of the program must be made with an appreciation of this essential fact.

The kind of general program design that has been described here, and its accompanying tools, are required not only because of the needs inherent in the fabrication of the first operational program, but also because of the inescapable fact that the specifics of the operational system and thus its program will demand changes as it is used and as changes take place in the operational environment. By now all programmers, whether they are working in the military, commercial, or scientific disciplines, or even for themselves, have become aware of the results of pursuing thoughts along the line of "Wouldn't it be nice if. . . ." Changes that seem simple from the point of view of external logic can be catastrophic in relation to the internal structure of the program, just because of the effect on the design decisions that we have briefly reviewed. Changes frequently affect many of the subprogram areas.

Therefore, those about to embark on a program design for a large on-line system should heed the following warning: Do not attempt such an enterprise with fewer automated aids for construction, checkout, and maintenance than are taken as a matter of course by the programmer of a small system. Add, as a corollary to this warning, that large real-time programs require new tools in

order to handle the production task adequately. This requirement is due partly to the problems of size and partly to the conflicting philosophies that exist in regard to storage allocation between the standard operating system and the translator of the programming language being used.

## SECTION IV

### CONSTRUCTION TOOLS USED FOR ON-LINE SYSTEMS

In order to isolate the produced code and thus the individual programmer from certain design decisions, three innovations (time period 1955-56) were made in the program production and the operational design of the SAGE System. The first of these was the use, for the operational program, of a table-driven central executive program to control the sequencing and I/O flow of the entire program. The tables involved were called the sequence parameters and were mainly a reflection of the changing but, from an on-line point of view, static design decisions. Dynamic conditionality indicators could be set for the operation of specific subprograms. This arrangement made it possible to effect, only when required, the transfer of appropriate operating environments for the conditional subprograms. In addition, certain tables could, within limits, be dynamic in size, and they were to be transferred and used with their actual size taken into account. The second innovation was the accumulation in a central bookkeeping table, referred to as the communication pool (COMPOOL), of certain descriptive information about the data and the subprograms of the operational system. The third innovation was the insertion into the operational system of a data- or storage-recording system that could, before or after any subprogram, record any of the storage regions of the system. This program was table-driven and set dynamically or before test operation. It could handle any reasonable amount of recording. Care must be taken in the use of such a recording system not to influence the very phenomena that are to be observed by changing the time of the system beyond reasonable bounds. This is especially important in operations using live inputs.



In order to code using the production tools available, the programmer of SAGE had to know what functions his subprogram had to perform, the system symbols of the data elements he required, whether or not the data was part of an array, the indexing quantity used for entry into the array, the coding (i. e., the specific numeric values and their meaning) of the data elements, and the symbol and entry parameters of any required system subroutines. He did not have to know the location of his program in core or backing store, the packing of the data elements, the association of the data elements into tables, or the location of the data elements in core or backing store--in fact, it was the intent of the program-production managers to withhold such information from the individual programmers.

The transformation of the symbolic code into machine code was a two-stage process. First, there was the compile or assembly stage, which produced, in addition to such things as a relative and allocatable binary code and a local symbol table, a list of subprogram requirements from a data-element and subroutine point of view. The data elements were listed according to whether they were used only, modified only, or both used and modified. Such a list not only helped verify programmers' statements of data and subroutine requirements but was very useful in checking out the overall system program.

The first stage of translation is sketched in Fig. 9. At this point, the only information needed from the COMPOOL is the relative location of each system symbol in a master list or lists.

The second stage of translation is normally called loading. At this point the programmer entered his output from the compiler, namely his symbol table and his relative binary deck. (In addition, he could add corrections in the program in terms of his local symbols and system symbols.) To perform this operation, the loader used the COMPOOL to determine the operating core

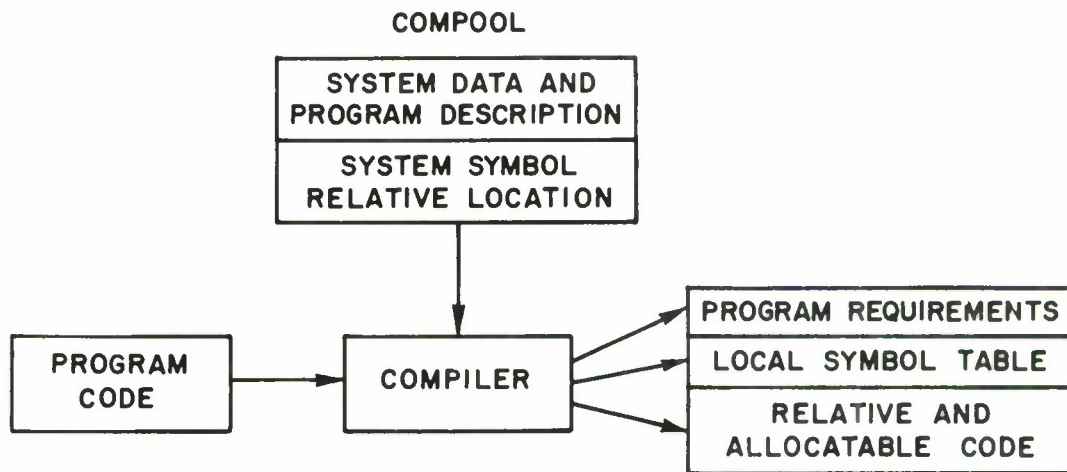


Fig. 9 First Stage of Translation--Compiling

address of the program, the location of the system subroutines called upon by the program for each data element referred to, the table in which it was located, the base address of that table in core, the mask needed to extract the data element from storage or deposit it into storage, the shift required to position the data element for testing or computation, and the shift required to restore the data element to the appropriate position for storing. After transforming the relative and allocatable code to absolute by the appropriate substitution of derived numbers, the loader would store the code in the appropriate place in backing store, leave it in its operating core position, or both. In addition, required system subroutines could be loaded.

It is apparent that most of the resource-allocation decisions that were found to be difficult to derive in the preliminary stages of program design are not used in the code translation until the load, or read-in, phase. Therefore,



such decisions can be made and changed without affecting the source code or the output of the first stage of translation, the compile phase. The loading process is shown in Fig. 10.

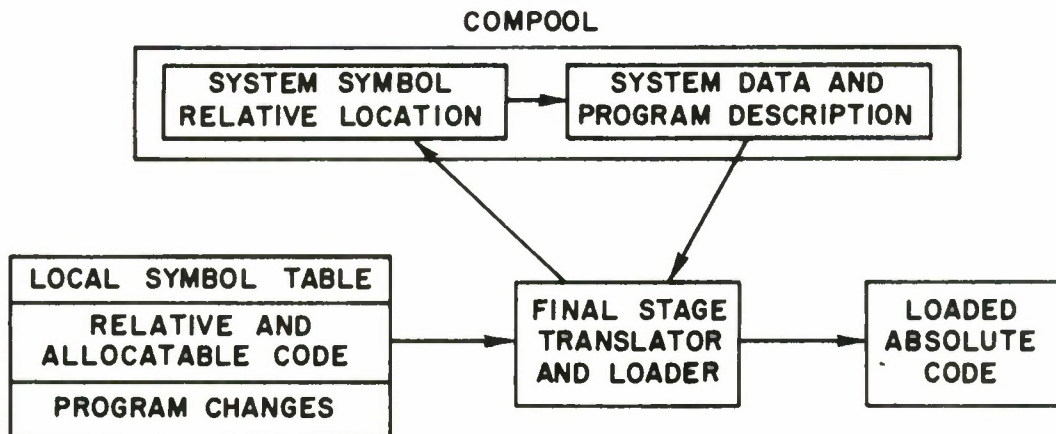


Fig. 10 Second Stage of Translation--Loading

During the initial phases of subprogram checkout it is not essential that the system data and program description in the COMPOOL be complete or up to date; it is necessary only that it be adequate for specific subprograms in checkout. There is no logical reason why there cannot be a separate description for each subprogram or why the one used cannot be inconsistent from an overall-system point of view if it is adequate for each subprogram. However, it would be difficult to check out a subprogram with the tools described so far, since there is no way to introduce values into the data elements referred to by a specific subprogram. Such a simple tool can easily be constructed. The information needed exists in the COMPOOL, and, therefore, simulation data can be produced for loading with the program, as illustrated in Fig. 11.

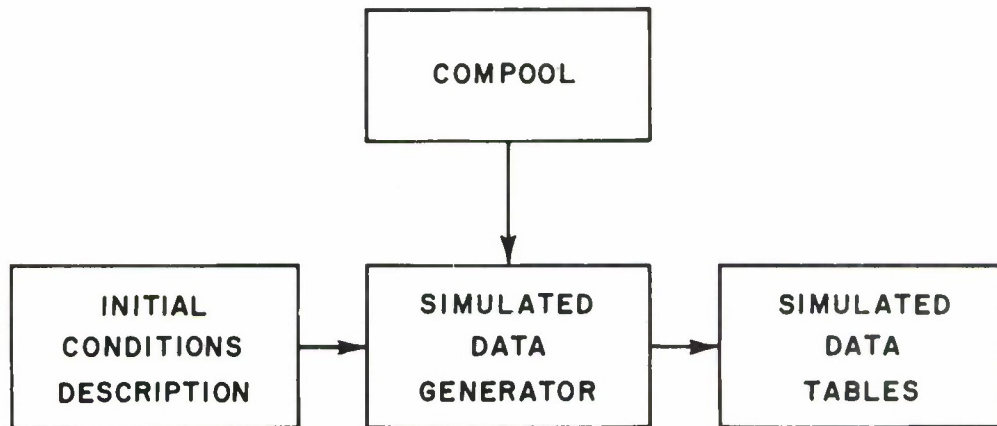


Fig. 11 Simulation-Data Production

In addition, it is necessary to produce a debug-control routine that accepts control statements in local- and/or system-form and controls code execution and recording during a checkout run. A flow diagram of the operation of such an execution is given in Fig. 12. The flow shown is idealized and does not represent any specific tool used in the production of SAGE, although it does characterize the effect of several tools used. Making the output of such a run available to the programmer would require a translator to change selected aspects of the recorded data into data-element values, local symbols, or system symbols for output to a printer. Such a phase is sketched in Fig. 13.

Such a debugging tool allows detailed checking of each subprogram. For system checking, or "assembly testing," as it is called in SAGE terms, new but similar procedures were used. In system tests, provision must be made for continuous or intermittent entry of exogenous data. Usually these data are

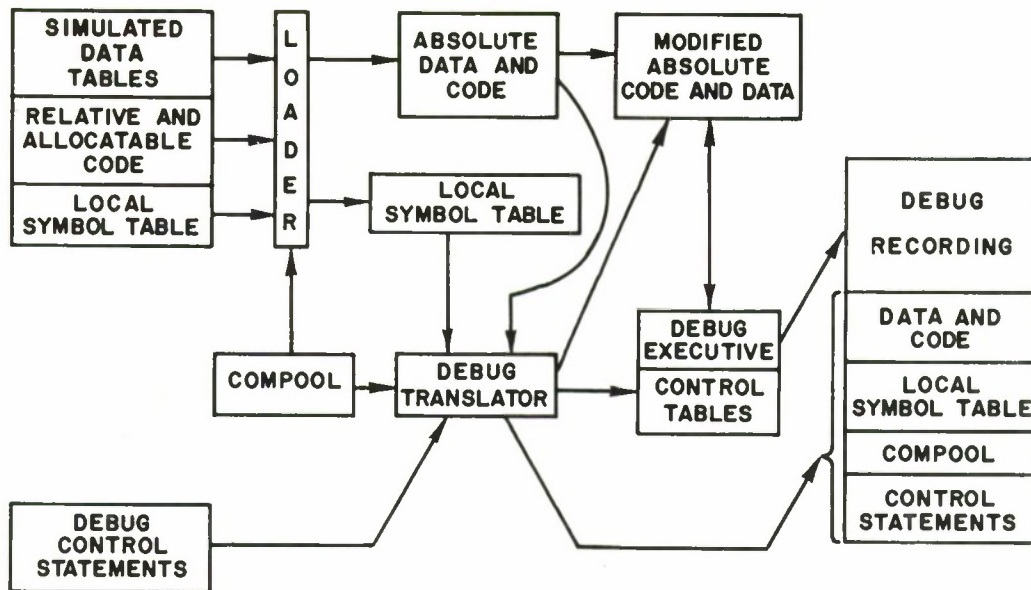


Fig. 12 Debug-Run Flow

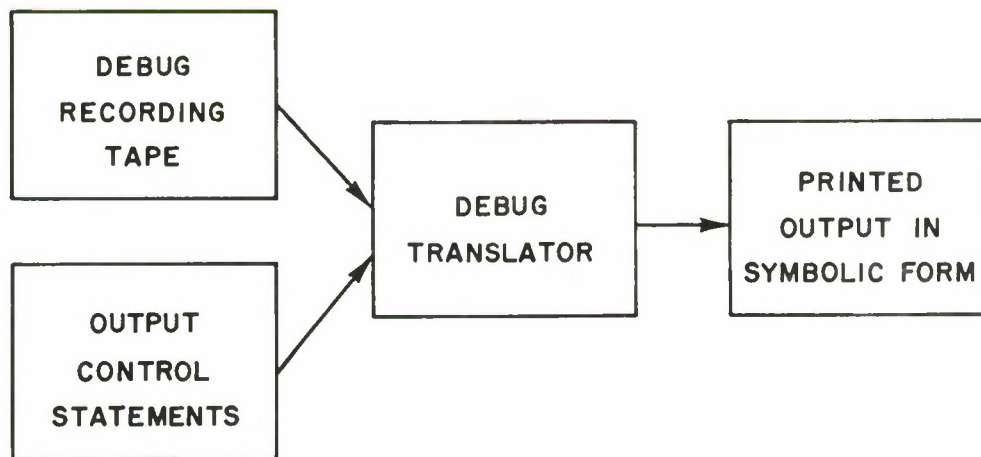


Fig. 13 Recorded-Data Translation

simulated and entered from tape, although real data can be used for noncontrolled but "realistic" inputs. This procedure is diagrammed in Fig. 14.

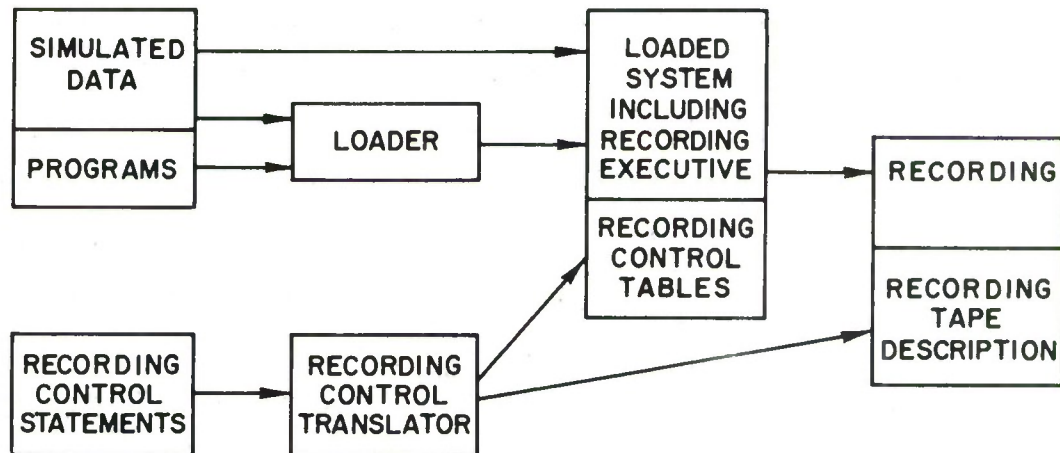


Fig..14 Assembly Testing

## SECTION V

### CONCLUSIONS

Some time has been spent here in reviewing the program-design process and the tools required to defer some critical design decisions. This is only to illustrate that the major design decisions have little relation to the programming language chosen for the statement of the logical data transformations to be implemented by the final program. The actual coding required for performing the logical transformations turns out to be a relatively small part of the overall design job. On the basis of his own experience, the author feels that the choice of the programming language to be used is just an initial decision in the process of selecting adequate tools for assisting the production and modification of real-time programs.

## DOCUMENT CONTROL DATA - R&amp;D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) The MITRE Corporation Bedford, Massachusetts		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3 REPORT TITLE Program Structure for Military Real-Time Systems			
4 DESCRIPTIVE NOTES (Type of report and inclusive dates) NA			
5 AUTHOR(S) (Last name, first name, initial) Burrows, James H.			
6. REPORT DATE January 1965	7a. TOTAL NO. OF PAGES 24	7b. NO. OF REFS 0	
8a. CONTRACT OR GRANT NO. AF19 (628)-2390	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TDR-64-161		
b. PROJECT NO. 416 and 502			
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)		
d.	SR-122		
10. AVAILABILITY/LIMITATION NOTICES Qualified requestors may obtain from DDC. DDC release to OTS authorize			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Directorate of Computers L. G. Hanscom Field Bedford, Massachusetts	
13. ABSTRACT This report describes the program design and accompanying tools required to fabricate an initial operational program which is subject to change through use and as a result of changes in the environment of the system. Major design decisions are shown to have little relation to the programming language that is chosen for the statement of the logical data transformations to be implemented by the final program. The actual coding required for performing the logical transformations turns out to be a relatively small part of the overall design job. The choice of the programming language to be used is just an initial decision in the process of selecting adequate tools for assisting the production and modification of real-time programs.			



14.	KEY WORDS	LINK A		LINK B		LINK C	
		ROLE	WT	ROLE	WT	ROLE	WT
Computers and Data Systems Data Processing Systems Programming (Computers)							

## INSTRUCTIONS

1. **ORIGINATING ACTIVITY:** Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization (*corporate author*) issuing the report.

2a. **REPORT SECURITY CLASSIFICATION:** Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. **GROUP:** Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. **REPORT TITLE:** Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. **DESCRIPTIVE NOTES:** If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. **AUTHOR(S):** Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. **REPORT DATE:** Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. **TOTAL NUMBER OF PAGES:** The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. **NUMBER OF REFERENCES:** Enter the total number of references cited in the report.

8a. **CONTRACT OR GRANT NUMBER:** If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. **PROJECT NUMBER:** Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. **ORIGINATOR'S REPORT NUMBER(S):** Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. **OTHER REPORT NUMBER(S):** If the report has been assigned any other report numbers (*either by the originator or by the sponsor*), also enter this number(s).

10. **AVAILABILITY/LIMITATION NOTICES:** Enter any limitations on further dissemination of the report, other than those

imposed by security classification, using standard statements such as:

- (1) "Qualified requesters may obtain copies of this report from DDC."
- (2) "Foreign announcement and dissemination of this report by DDC is not authorized."
- (3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through \_\_\_\_\_."
- (4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through \_\_\_\_\_."
- (5) "All distribution of this report is controlled. Qualified DDC users shall request through \_\_\_\_\_."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. **SUPPLEMENTARY NOTES:** Use for additional explanatory notes.

12. **SPONSORING MILITARY ACTIVITY:** Enter the name of the departmental project office or laboratory sponsoring (*paying for*) the research and development. Include address.

13. **ABSTRACT:** Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as (TS), (S), (C), or (U).

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. **KEY WORDS:** Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, roles, and weights is optional.